

SWE599

PROJECT REPORT - DRAFT

**SPATE - Simulated Poker
Analysis and Test Environment**

Author:

Deniz DIZMAN

May 30, 2010

1 Introduction

Computer based poker agents are a current hot research topic with many universities producing publications and implementations about their advances on the subject like the University of Alberta which produced Poki and Vexbot [2] [1]. Also many hobbyist computer programmers are implementing different agents and are constantly arguing on forums (such as <http://www.pokerai.com>) which approach to implementing the perfect player is better. There are many different approaches that can be taken when designing a poker playing program such as game theoretic methods, game tree search methods, Bayesian methods, etc. and it is not always easy to measure the competence of the program with regard to other players. Spate is a platform and API that tries to eliminate this handicap by providing agents with a poker playing environment and a detailed trace of the game later analysis and measurements. The document will explain the architecture of the Spate system.

2 Botpit requirements

2.1 Introduction

The SPATE Botpit tool is a poker game governor application that currently supports simulating chas games of limit Texas Hold'em poker. Agents that implement the Spate API are loaded dynamically by the application and the engine will start to simulate the specified amount of poker games between the agents. Spate will marshal these attributes of the game:

1. Bets
2. Actions (call, raise, fold) in turn
3. Blinds
4. Game Stages (preflop, flop, turn, river)
5. Winners for the hands
6. Distribution of the pot to winner(s)

All of the attributes are logged to a log file that can be later used by an analysis tool for performance measurements of the agents and visual replay of the game. Figure 1 depicts an overview of Spate

Agent: Agents are external Java Jar archives that contain the agent class files that must implement the interface provided by Spate. The simulation engine will query the agents in turn for a specific action and will provide the agents with information regarding the game status, table status or other agents statuses.

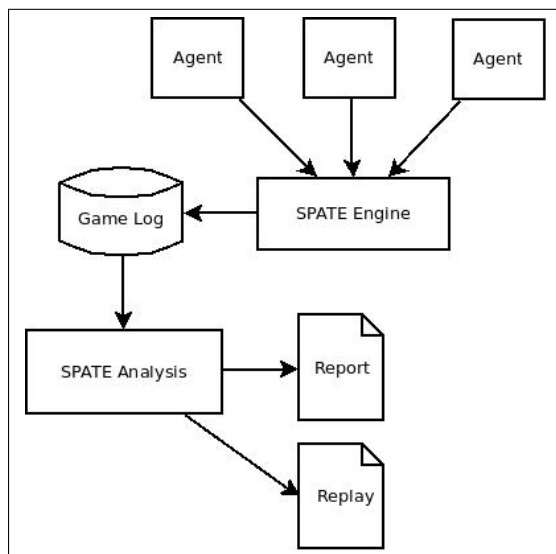


Figure 1: SPATE architecture

2.2 Functional requirements

1. Dynamic loading of external agents: The system must be capable of loading external agents that adhere to a specific interface. The agents to be loaded must be packed as a standard Java JAR archive. Their file names shall be specified in the configuration file and the agents shall be loaded in the order they are found in the configuration file. If any of the loaded JAR files does not implement the required interface the loader will warn the user about the error and shall not run the simulation.
2. Stack sizes for all players must be definable by the user: The user must be able to define different stack sizes in the configuration file for the simulation round. Each player will start with this amount of money and through out the simulation no re-buys are permitted.
3. The table blinds (big and small blind) must be definable by the user: The user must be able to define the small and blind sizes in the configuration file. The simulation will then proceed with these blinds. The blinds will remain constant through out the simulation.
4. Logging levels must be adjustable: The user shall be able to define the log level of the application from the configuration file. The allowed log levels are INFO and DEBUG. The DEBUG log level shall output verbose information about the table state and application state to aid in debugging the application. The INFO level shall be less verbose and only inform the user about important events during the simulation.

5. The number of simulation rounds must be definable by the user: The user must be able to define how many hands the simulation will consist of. Exactly that number of hands will be played during the simulation.
6. The game tracer should be on/off switchable: The application shall keep a separate trace log of the game which will be used for post game analysis by the Spade analysis tool. This log file must contain all relevant information of the game and players for the stats the analysis tool will generate. Format for the log file is defined in the relevant section of this document.
7. The full rule set of the game Limit Texas Hold'em poker should be implemented: The simulation shall adhere to the rules of sit and go limit Texas Hold'em poker. The betting structure (blinds, raise threshold), hand rankings, allowed actions shall be enforced by the application.
8. Games that end in draws should be taken into consideration: The application should divide the pot equally among any players that have the same hand rank at the end of a hand. Otherwise the winning player should collect the pot.

2.3 Non-functional requirements

1. Minimum of 100K Hands per minute independent of bots: For the maximum amount of agents allowed (10) the application should be able run at least 100K hands per minute given that the agents take a negligible amount of time to make a poker decision.
2. Support for windows XP and later, Linux 2.6.24 and later, Mac OS X and later: The application should be runnable under the mentioned operating system that provides the Java run time environment version 5 or higher.

3 Botpit design

3.1 Botpit Modules

3.1.1 Agent Loader and interface

The Agent loader will load external agent JAR files from the disk on the classpath and place them in the relevant table slots (seats). The configuration file holds the agent implementations to be loaded in the format:

```
player.<Seat Number> = <Agent Implementation fully qualified class name>
```

For example to load 3 players of the same type (CallBot) the configuration file would contain the following lines:

```
player.1 = com.swe599.spades.bots.CallBot
player.2 = com.swe599.spades.bots.CallBot
player.3 = com.swe599.spades.bots.CallBot
```

3.1.2 Game Tracer Module

The game tracer module logs all of the core aspects of the game to a file for later analysis. To increase the performance of the tracer a buffer of N lines are allocated in a String array and trace log entries are recorded into this array. The buffer is dumped to the file when the array is full and the array pointer is reset to point the beginning of the array. The elements in the array are not cleared and overwritten by new entries to increase the performance of the trace logger. The tracer can be turned on or off by setting the configuration value `tracing = true` or `tracing = false` in the configuration file `botpit.properties`.

The format of the trace log is defined as follows

`<id>. <action_type> <message>`

1. id: is a unique marker for the log entry.
2. action type: specifies the action that the log entry describes.
3. Message: the actual information provided by the log entry

Here is an excerpt of the trace log from a simulation:

```
00000000. [hand] id: 0
00000001. [players] total: 3
00000002. [stage] preflop
00000003. [action] seat: 2 type: post_smallblind
00000004. [action] seat: 0 type: post_bigblind
00000005. [stack] seat: 2 amount: 4999.000000
00000006. [stack] seat: 0 amount: 4998.000000
00000007. [bet] seat: 2 amount: 1.000000
00000008. [bet] seat: 0 amount: 2.000000
00000009. [pot] amount: 3.000000
00000010. [holecard] seat: 0 c1: As c2: 3d
00000011. [holecard] seat: 1 c1: 6s c2: 4c
00000012. [holecard] seat: 2 c1: 5c c2: 5s
00000013. [action] seat: 1 type: fold
00000014. [stack] seat: 1 amount: 5000.000000
00000015. [bet] seat: 1 amount: 0.000000
00000016. [pot] amount: 3.000000
00000017. [action] seat: 2 type: call
00000018. [stack] seat: 2 amount: 4998.000000
00000019. [bet] seat: 2 amount: 2.000000
00000020. [pot] amount: 4.000000
00000021. [action] seat: 0 type: check
00000022. [stack] seat: 0 amount: 4998.000000
00000023. [bet] seat: 0 amount: 2.000000
00000024. [pot] amount: 4.000000
00000025. [stage] flop
```

```

00000026. [board] 4d Th 7c __ __
00000027. [action] seat: 2 type: check
00000028. [stack] seat: 2 amount: 4998.000000
00000029. [bet] seat: 2 amount: 0.000000
00000030. [pot] amount: 4.000000
00000031. [action] seat: 0 type: check
00000032. [stack] seat: 0 amount: 4998.000000
00000033. [bet] seat: 0 amount: 0.000000
00000034. [pot] amount: 4.000000
00000035. [stage] turn
00000036. [board] 4d Th 7c 2d __
00000037. [action] seat: 2 type: check
00000038. [stack] seat: 2 amount: 4998.000000
00000039. [bet] seat: 2 amount: 0.000000
00000040. [pot] amount: 4.000000
00000041. [action] seat: 0 type: check
00000042. [stack] seat: 0 amount: 4998.000000
00000043. [bet] seat: 0 amount: 0.000000
00000044. [pot] amount: 4.000000
00000045. [stage] river
00000046. [board] 4d Th 7c 2d 8d
00000047. [action] seat: 2 type: check
00000048. [stack] seat: 2 amount: 4998.000000
00000049. [bet] seat: 2 amount: 0.000000
00000050. [pot] amount: 4.000000
00000051. [action] seat: 0 type: check
00000052. [stack] seat: 0 amount: 4998.000000
00000053. [bet] seat: 0 amount: 0.000000
00000054. [pot] amount: 4.000000
00000055. [stage] showdown
00000056. [winner] seat: 2 amount: 4.000000
00000057. [endstack] seat: 0 stack: 4998.000000
00000058. [endstack] seat: 1 stack: 5000.000000
00000059. [endstack] seat: 2 stack: 5002.000000
00000060. [endhand] id: 0

```

The trace log contains all information for every agent and the table.

3.1.3 Configuration Module

The configuration module loads the configuration file named `botpit.properties` from the class path and assigns game parameters from the configuration to the simulation engine. It also loads agent definitions. The format of the configuration file is as follows

```
<configuration_key> = <configuration_value>
```

The configurable parameters are the following:

Parameter name	Type	Description
bbsize	float	the size of the big blind bet
maxraises	int	number of maximum raises
sbsize	float	the size of the small blind bet
logging	bool	application debug logging
numrounds	int	number of simulations
numplayers	int	number of players
trace	bool	game trace logging

3.1.4 Simulation Engine

The simulation engine runs and governs the game between the agents. Each game begins with a new hand and the engine clears board cards, sets the pot to zero, sets the raises to zero increments the dealer button to point the next player, shuffles the deck and informs the agents that a new hand has begun. Then the simulation proceeds by the engine simulating the game stages pre-flop, flop, turn, river and showdown. Should the game not commence to a flop stage the engine rewards the pot to the winning player. In the preflop stage of the game the blinds are deducted from the players in the two slots left to the dealer and added to the pot. The cards are dealt to the agents or the board according to the game stage. The simulation engine then checks if a player had already folded, and if not asks for the current player's action. The engine then interprets the action and saves it to the action table for further reference. Until the betting stage is over each agent is asked for its action and accordingly any bets are deducted from the agents stack and added to the pot. The condition to decide if a betting is over is determined by the engine by checking if every that has not folded, has matched the maximum bet. At each stage except from the showdown stage the engine checks if every player but one has folded, which means that there will be no showdown stage, and the pot is awarded to the this player and the hand is over. At the showdown stage the pocket cards from the agents still in the game are are appended to the 5 board cards and the SpateEval library's 7 card hand evaluator is called to determine a ranking. The agent owning the top ranking hand is rewarded the pot and the a new hand begins. In the case of 2 or more agents that have the same rank, the pot is split equally among them.

3.1.5 Graph Module

The graph module shows a real-time graphic view of the agents current stacks during the simulation. The graph module will query the simulation engine every

second to reflect the updates of the stacks. The graph view is optional and can be switched on by supplying the “-g” parameter on the command line to the application.

3.2 Botpit Architecture Overview

The system architecture is a call-and-return based architecture where the main Botpit class will call the relevant methods of its self and its accompanying helper class in a sequential manner.

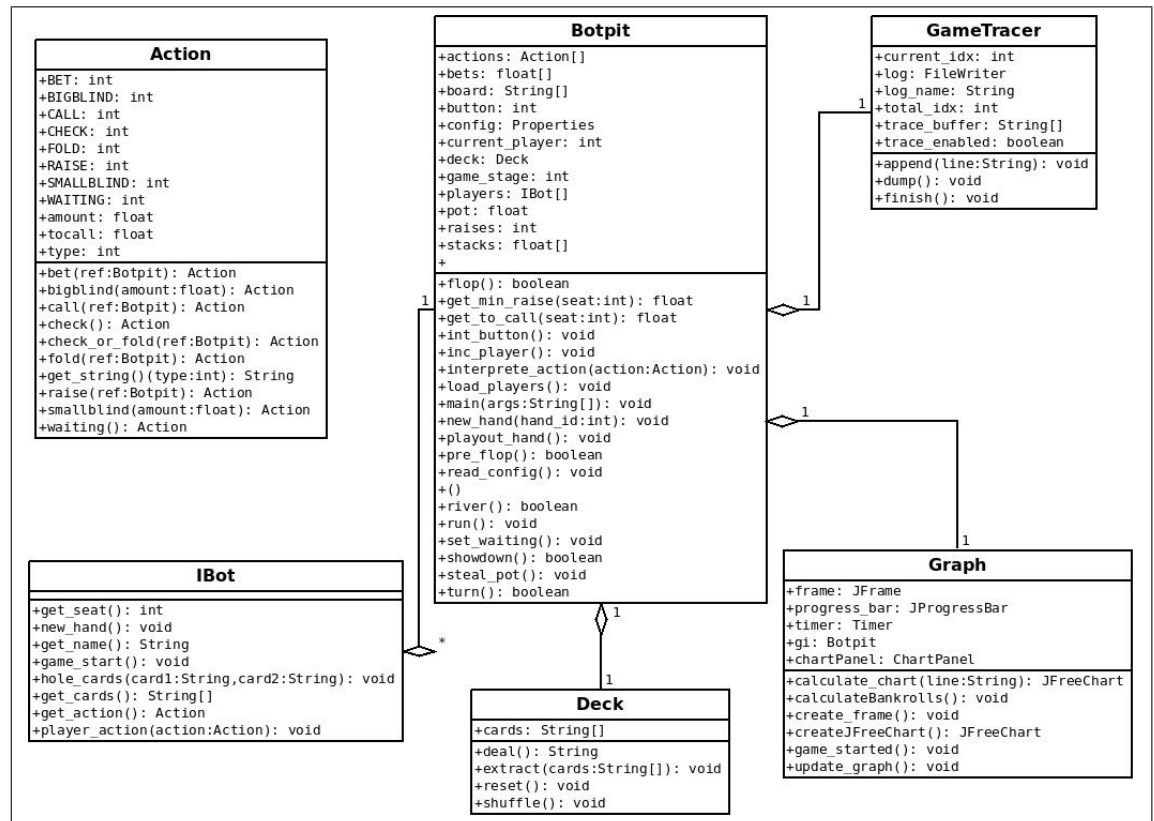


Figure 2: Botpit class diagram

3.2.1 Graph Class

This class represents a real time graphical view of the stacks of the agents during the simulation. This view can be turned on or off by specifying the command line argument “-g” when launching the application. The code in this class is a simplified version of [3] using JFreeChart [6]

Method Descriptions

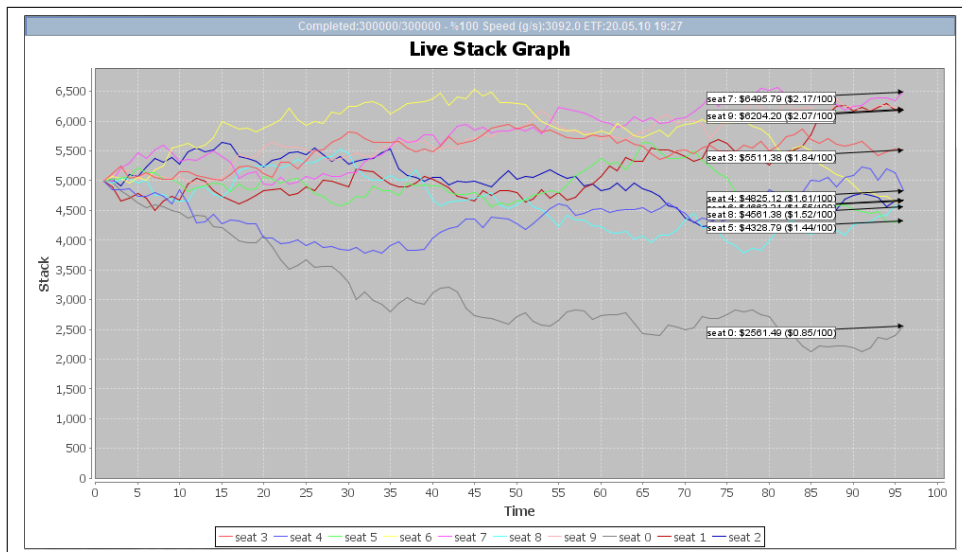


Figure 3: Screenshot of the graph view

`calculate_chart()`

Calculates game speed and estimated time of finish for the graph

`calculateBankRolls()`

Calculates the player stacks for the graph

`create_frame()`

Creates the graphical user interface with the progress bar and chart

`createJfreeChart()`

Creates the and displays the chart.

`game_started()`

Notifies the graph module that the simulation has started.

`update_graph()`

Creates a timer that will periodically execute in a separate thread and update the chart.

3.2.2 Game Tracer class

This class records all of the events in the game to generate the trace log for later analysis. Method Descriptions:

`append()`

Appends the provided line to the log buffer. All the lines to be logged in the trace log are held in an array, with an array index pointer pointing to the next slot for the next line. Each time a new line is appended the pointer is incremented by to point to the next slot. After the line array is full the data is dumped to the log file and the pointer is reset to the first element. The array is not cleared to preserve CPU cycles.

`dump()`

Appends the log buffer to the log file.

`finish()`

Saves and closes the log file. Only the elements up to the array index pointer are saved. This method ensures that dirty elements in the line array are not duplicated in the log file when the contents is written.

3.2.3 Deck class

This class represents a standard deck of 52 cards as an integer array consisting of integers from 0 to 51. Method Descriptions:

`deal()`

Deal one card from the top of the deck. The deck class holds a pointer that points to the top of the deck. The card in the cards array at the pointer position is returned.

`extract()`

Extracts the given cards from the deck.

`reset()`

Resets the deck of cards to the beginning state by restoring extracted cards and undoing any shuffles. The deck class contains a hard coded 52 card deck array. To increase performance this array is copied to the card array. The current card pointer is reset to zero.

`shuffle()`

Shuffles the deck by select 2 random cards and swapping their places in the deck.

3.2.4 Botpit class

This class is the simulation engine and consist of the major functionality modules. Method Descriptions

`flop()`

Runs the game stage flop

`get_min_raise()`

returns the minimum amount that the player may raise taking into account the number of current raises.

`get_to_call()`

returns the amount for an agent to call the current bet

`inc_button()`

set the dealer button to the next player on the table

`inc_player()`

advances the pointer that holds the next player to act

`interpret_action()`

Interprets the agents action for the current game stage.

`load_players()`

Loads the players from the provided agent classes and configuration file.

`main()`

Entry point for the application

`new_hand()`

begins a new hand.

`playout_hand()`

plays out the hand for the current stage. Asks for agent actions and governs betting.

`pre_flop()`

Runs the preflop game stage

`read_config()`

Reads the configuration file from disk.

`river()`

runs the river stage of the game.

`run()`

runs all of the game stages and checks if the game terminates without a showdown.

`set_waiting()`

sets the players action to waiting

`showdown()`

runs the showdown stage, compares players hands and determines the winner of the hand. Uses the SpadeLib hand evaluator for hand ranks. Splits the pot evenly between winning players should there be more than one winner.

`steal_pot()`

If the game terminates without reaching a showdown, awards the pot to the player still in the game.

`turn()`

Runs the turn stage of the game.

3.2.5 Action class

This class represents each action that a poker agent may take during the game. It is a static class because each agent refers to the same actions but with different parameters. Method Descriptions:

`bet()`

This method returns an action type of bet. The bet amount is taken from the Botpit reference passed as a pointer.

`bigblind()`

This method returns a big blind post action. The amount is taken from the Botpit reference.

`call()`

This method returns a call action.

`check()`

This method returns a check action

`check_or_fold()`

This method returns a check action if no betting has been performed. Otherwise is returns a fold action.

`get_string()`

This method returns the string representation of the action

`raise()`

Returns an action of type raise

`smallblind()`

returns a small blind action

`waiting()`

returns an action that indicates that the player is waiting to perform an action

3.2.6 IBot class

This class is an interface that all agents must implement in order to be able to play at the table. This interface is based a simplified version of [2] Method Descriptions:

`get_seat()`

Returns the seat number of the agent.

`new_hand()`

Informs the agent that a new hand has begun

`get_name()`

Returns a string that represents the agents name

`hole_cards()`

Informs the player of its hole cards.

`get_cards()`

Returns the players hole cards.

`get_action()`

Returns the players action for that hand

`player_action()`

informs an agent of the action of the other agents

4 Botpit Performance

4.1 Introduction

Today there are different test platforms such as [3] [4]. Different applications have different priorities, but one priority that must be common is performance. SPATE was built from the ground up with performance in mind.

4.2 Comparisons

As in comparison to SPATE the referenced platforms were chosen. To make a sound comparison the chosen applications must all implement the Meerkat API [4] and be opensource. The following simulations were run 3 times on a AMD 2600 XP work station with 8GB RAM with the JVM parameters set to -server -Xms128m -Xmx2048 and 10 agents that only issue the Call Action (which has a negligible call time) and the mean values were taken.

Table 1: Game tracing disabled

Program	Hands/sec	Memory(mb)	Notes
SNG Public test bed	426	63	25K Hands
Opentestbed	5002	77	100K Hands
Spate	45710	246	300K Hands

SNG public test was tested with 25K hands due to time restrictions. Opentestbed cannot simulate more than 100K hands. From the results it can be seen that spate is 9 times faster than its closest competition opentestbed when the game tracing is disabled.

Table 2: Game tracing enabled

Program	Hands/sec	Memory(mb)	Notes
SNG Public test bed	298	64	25K Hands
Opentestbed	4545	77	100K Hands
Spate	4101	278	300K Hands

The results show that with game trace enabled open test is a %10 faster than Spate. The reason for this is that the game trace log generated by spate is 1.5 times larger than opentestbed. Opentestbed records the log in Full tilt poker compatible format, whilst Spate has a custom format, that is much more detailed than FTP format. This amount of time spent on doing file input/output naturally reflects on the results. Full tilt poker format support in Spate is planned in the future.

4.3 Profiling

Below is a table representing the profiling data gathered by the Eclipse Performance Tools Platform Project (<http://www.eclipse.org/tptp/>) during a 100 hand simulation with 10 agents all Call bots.

Method	Base time	Avg Base time	Cumulative Time	# of calls
Botpit	13.814965	0.000603	28.213958	22906
payout_hand()	7.134471	0.017836	21.245583	400
is_betting_over()	1.133713	0.000283	1.724146	4000
is_noshowdown()	0.65991	0.000147	0.65991	4500
get_to_call()	0.642855	0.000146	0.642855	4400
set_waiting()	0.592507	0.001481	2.16637	400
interpret_action()	0.571421	0.000159	0.571421	3600
new_hand()	0.570894	0.005709	1.449381	100
pre_flop()	0.550356	0.005504	6.437677	100
inc_player()	0.499722	0.000139	0.499722	3600
showdown()	0.495955	0.00496	3.455211	100

The data from the table shows that the most time consuming methods during the simulation are – payout hand method is not taken into account because it is the calling method for all other methods – pre_flop(), showdown() and set_waiting() which are candidates for further optimization. The showdown method is of special importance due to the fact that hand ranking algorithms are run in this method.

5 Conclusion

Spate was designed and implemented with speed in mind because in order to reduce variance between different agents and to obtain good metrics an abundant amount of games must be simulated. In order to achieve this a flexibility and maintenance trade-off was chosen. Interpreting the results of benchmarks Spate seems to have achieved what it has promised and is ready to be used by the poker botting community.

References

- [1] Billings, D.; Davidson, A.; Holte, R.; Schaeffer, J.; Schauenberg, T.; and Szafron, D.; Bowling M. *Game tree search with adaptation in stochastic games of imperfect information* University of Alberta University, 2006.
- [2] Billings, D.; *Algorithms and assessment in poker* Phd. Thesis, University of Alberta

- [3] Schatzberg, Dan. *Opentestbed Open Meerkat Poker testbed*
<http://code.google.com/p/opentestbed/>
- [4] Poker Academy <http://www.poker-academy.com>
- [5] Web posting on 2p2 forums. <http://archives1.twoplustwo.com/showflat.php?Cat=0Number=8513906page=0>
- [6] JFreeChart <http://www.jfreechart.org>